DETECTION OF CONTROL FLOW ERRORS IN PARALLEL PROGRAMS AT COMPILE TIME

Bruce P. Lester

Department of Computer Science
Maharishi University of Management, Fairfield, Iowa, USA
blester@mum.edu

ABSTRACT

This paper describes a general technique to identify control flow errors in parallel programs, which can be automated into a compiler. The compiler builds a system of linear equations that describes the global control flow of the whole program. Solving these equations using standard techniques of linear algebra can locate a wide range of control flow bugs at compile time. This paper also describes an implementation of this control flow analysis technique in a prototype compiler for a well-known parallel programming language. In contrast to previous research in automated parallel program analysis, our technique is efficient for large programs, and does not limit the range of language features.

KEYWORDS

parallel programming; automated verification; parallel compilers; deadlock; control flow graph; frequency analysis

1. INTRODUCTION

One of the major problems in the field of parallel computing is the difficulty of software development. Parallel programs are inherently more complex than sequential programs, and therefore require significantly more time to write and debug. Also, the execution of parallel programs is fundamentally nondeterministic in nature, making it even more difficult to locate program errors through the standard testing and debugging techniques used for sequential programs.

Any automated technique for helping to detect errors in parallel programs could be very valuable to aid in parallel software development. In this paper, we give an overview of one such promising technique: analysis of global control flow properties of a parallel program using a system of linear equations. This system of equations, derived from what we call the *flow matrix* of the program, can be solved using standard methods of linear algebra to detect control flow errors in the overall structure of a parallel program. This technique can be used by a parallel program compiler or debugging system to help the programmer locate structural errors in the program.

Over the years, a considerable amount of research has been published on techniques for automated debugging of concurrent programs. However, these techniques have not seen widespread practical application because each technique suffers from one or more of the following problems:

1. The computational complexity of the technique is too great for application to large programs.

2. To apply the technique to a real programming language, some of the commonly used language features must be eliminated.

3. The technique cannot be completely automated in a compiler. The programmer must create an abstract model of the program for analysis purposes.

The *flow matrix* technique presented in this paper does not suffer from any of these three problems. The technique is computationally tractable: the worst case time complexity is O(n^3), with a wide range of parallel programs requiring only O(n) time. The technique can be automatically applied by a compiler to parallel programs that use the full range of language features. We have already developed a prototype compiler that analyzes real programs written in a well-known parallel programming language.

The most common type of automated analysis technique found in the research literature is the *finite-state verification* technique. Typically, these techniques generate a reachability graph of all possible control states of the parallel program, then search the graph for pathological states, such as program deadlock. Since the number of reachable states is usually an exponential function of program size, these techniques focus on reducing the size of the reachability graph. Some examples can be found in Avrunin [1, 2], Godefroid and Wolper [3], Holzmann [4], Helmbold and McDowell [5], and Valmari [6].

Avrunin [7] gives a good survey of finite-state verification techniques, and compares the performance of four of these techniques on a particular concurrent program. None of these techniques can be applied directly to the program source code, and all require the programmer to create a separate formal description of the program behavior. Whereas, our *flow matrix* technique is applied by a compiler directly to the parallel program source code.

Much of the previous research has focused on deadlock analysis of concurrent tasking in the ADA programming language (e.g. Masticola and Ryder [8] and Schatz [9]). To reduce the number of reachable control states, the analyzed programs are usually limited to simple process synchronization for controlling access to shared resources. In contrast to this, the control flow analysis technique presented in our paper is intended for parallel programs written for multicore (or multiprocessor) computers. The interprocess communication in such parallel programs often creates such a large number of reachable control states, that the finite-state verification techniques presented by other researchers are just not practical.

Flow graph techniques for detecting potential faults in parallel programs are continuing to be explored by other researchers. Schaeli and Hersch [10] use heuristics and previous execution information to reduce the state space of the flow graph for more efficient analysis. Baah, Podgurski, and Harrold [11] use estimates of program branch probabilities using a set of test inputs. Elkarablieh, Marinov, and Khurshid [12] use structural constraints to reduce the program state space for more efficient search. The *Racer* system presented by Bodden and Havelund [13] requires the programmer to add special primitives to the program to assist in the automated analysis. In contrast to these flow graph techniques, our *flow matrix* technique does not require any additional information beyond the source code of the parallel program itself.

Some automated analysis techniques rely on dynamic analysis while the program is running or using statistics gathered during program execution. Two examples of recent research efforts in this area are Tian, et al. [14] and Chen and MacDonald [15]. Whereas, our technique is done completely at compile time without any need to actually execute the program. Our paper is an extension of an earlier summary paper we presented at the International Conference on Parallel and Distributed Computing and Systems [16]. This new paper presents significant new research findings and extends the earlier paper to the C programming language.

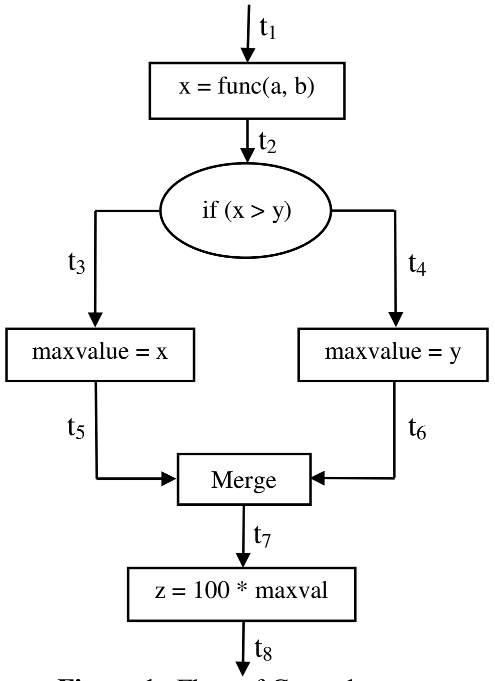

Figure 1. Flow of Control.

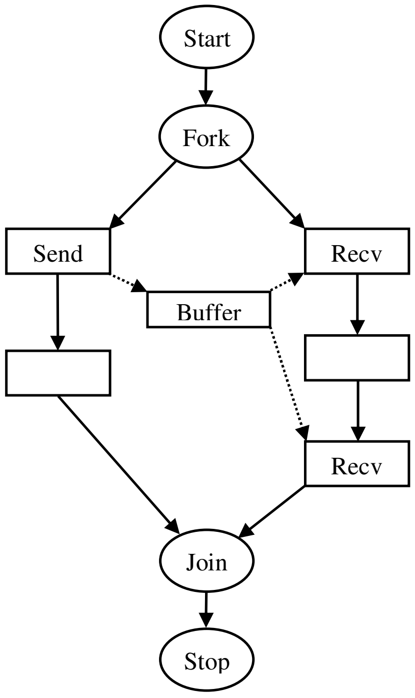

Figure 2. Control Flow Error in Parallel Program

2. Control Flow Graphs

It is well known that the execution frequencies of instructions in a computer program have a certain simple relationship. For example, consider the following fragment of a C program (the lines are numbered L1 – L5 for reference):

```
L1  x = func(a, b);
L2  if (x > y)
L3     maxval = x;
L4  else maxval = y;
L5  z = 100 * maxval;
```

For line L1, let [L1] denote the total number of executions of line L1 during a given execution of the whole program. Let us call [L1] the *count* for line L1. Clearly, [L1] = [L2] = [L3] + [L4] = [L5]. This relationship of the counts is seen even more clearly in the flowchart representation of these instructions as shown in Figure 1. A flowchart is a directed graph having an instruction associated with each vertex of the graph. The edges of the graph represent the paths for flow of control between the instructions. During execution of the flowchart, the flow of control can be represented by the movement of a control token (the execution point). For a given edge *t*, we use [t] to denote the total number of control tokens that have passed through that edge. [t] is called the *count* for edge *t*. Obviously, the following system of linear equations must hold for any properly terminating execution of the flowchart shown in Figure 1:

$[t_1] - [t_2] = 0$

$[t_2] - ([t_3] + [t_4]) = 0$

$[t_3] - [t_5] = 0$

$[t_4] - [t_6] = 0$

$[t_5] + [t_6] - [t_7] = 0$

$[t_7] - [t_8] = 0$

Using the standard methods of linear algebra, these *control flow equations* can be solved to provide useful information about the relationship between the execution frequencies of the various instructions in the program. All of this is well known for ordinary sequential computer programs. However, these control flow equations become much more interesting for parallel programs because the equations sometimes do not have a solution. Furthermore, the lack of a solution will sometimes indicate the presence of control flow bugs in the parallel program.

The flowchart form of an example parallel program is shown in Figure 2. A *Fork* operation creates two parallel processes that communicate using *Send* and *Recv* operations through a shared message buffer. The process on the right branch has two *Recv* operations, but the process on the left has only one *Send*. Assuming that *Recv* is blocking (i.e. it waits indefinitely for a message to arrive), this parallel program will result in an improper termination, sometimes called a deadlock. As was done for the sequential program in Figure 1, it is possible to determine the control flow equations for the parallel program of Figure 2. However, these control flow equations have no solution, which indicates an error in the flow of control in the parallel program. After this somewhat informal introduction, we will now give a more formal definition of control flow graphs, and describe how the resultant control flow equations can be used to detect control flow bugs in parallel programs.

A *control flow grap*h is defined as a finite, connected, directed graph. Associated with each vertex in the graph is a module that places local restrictions on the flow of control, expressed as a set of linear equations. For each module, its incoming edges are called *input terminals*, and its outgoing edges, *output terminals*. Control flows from one module to another by the movement of discrete *control tokens* along the terminals. The terminals do not have any capacity for storing control tokens. They just provide a path for movement of control tokens between modules. For a given terminal *t*, the *count* (denoted [t]) is defined as the total number of control tokens that have passed through that terminal since the start of the parallel program execution. Obviously, [t] will always be a nonnegative integer.

Each module has two control states: *active* and *inactive*. (Informally, we may think of an active module as one that is in the middle of an execution.) Initially, all modules are in the inactive state. A control flow graph has exactly one *START* module with two terminals: an input terminal and an output terminal. This single *START* module combines the functions ordinarily associated with separate *START* and *STOP* modules. When in the inactive state, the *START* module outputs a single control token through its output terminal and enters the active state. Once in the active state, it will accept a single control token through its input terminal and then enter the inactive state. If all the other modules are also in the inactive state at this time, the control flow graph is said to have reached a *proper termination*.

Definition: A control flow graph is defined as *properly terminating* if for each terminal *t*, there exists a properly terminating execution of the graph for which $[t] > 0$.

Notice that the above definition allows the possibility that some portion of the program may not be executed during a specific terminating computation. However, for the control flow graph to be considered as *properly terminating*, every portion of the program must be executed in some properly terminating computation.

Whenever a module is in the *inactive* state, the counts at its input and output terminals are constrained to satisfy a specific system of linear equations. More formally, each module *M* has an associated *control flow matrix* F_M of rational numbers. If module *M* has *n* terminals, then the flow matrix F_M has m rows and n columns. Throughout the execution history of the control flow graph, whenever module M is in the *inactive* state, the counts at its terminals are constrained to satisfy the following system of linear equations:

$$F_M\, T = \mathbf{0}$$

where T denotes the n-element vector of terminal counts ($[t_1]\ [t_2]\ \ldots\ [t_n]$) for module M.

Row i of the above system of linear equations will have the following general form:

$$f_{i1}[t_1]\ +\ f_{i2}[t_2]\ +\ \ldots\ +\ f_{in}[t_n] = 0$$

These linear equations for the individual modules can be simply combined in the obvious way to create a system of linear equations and corresponding control flow matrix F for the whole control flow graph. Consider the following system of linear equations:

$$F\,\boldsymbol{x}\ =\ \mathbf{0}$$

where $\boldsymbol{x}$ is the vector ($x_1, x_2, \ldots, x_n$), and $\mathbf{0}$ is the n-vector of zeroes.

Definition: A control flow graph is defined as *consistent* if the above system of linear equations has a solution in which $x_i \neq 0$, for all i.

We have now defined two possible properties of a control flow graph: *properly terminating* and *consistent*. *Properly terminating* is a behavioral property based on valid executions of the program. Whereas, *consistency* is a mathematical property based on the linear equations associated with the modules. The following theorem relates these two properties.

Theorem: Any control flow graph that is not *consistent* is not *properly terminating*.

Proof: The proof follows easily from the definitions. Assume that a control flow graph P with control flow matrix F is properly terminating. For any given terminal t, there must be a properly terminating execution of P for which $[t] > 0$. Since all modules must be in the *inactive* state after any properly terminating execution, the terminal counts for this execution must satisfy the linear equations associated with each module. Therefore, the terminal counts for this execution provide a nontrivial solution to $F\,\boldsymbol{x}\ =\ \mathbf{0}$, in which $x_t > 0$.

For each terminal t in P, there exists such a solution. We simply sum these solutions to get another solution to $F\,\boldsymbol{x} = \mathbf{0}$ in which $x_t > 0$, for all t. Therefore, P is *consistent*. We have shown that P is *properly terminating* implies that P is *consistent*. Therefore, if P is not *consistent*, then P is not *properly terminating*.

(End of Proof)

The number of rows and columns in the flow matrix F is proportional to the number of modules n in the control flow graph. Using standard techniques of linear algebra [17] to solve $F\,\boldsymbol{x}\ =\ \mathbf{0}$, the worst case time-complexity for determining consistency is $O(n^3)$. As we will see, the flow matrix F is usually very sparse, with only two or three nonzero entries in each row or column. Thus, sparse matrix techniques [18] can be applied to reduce the time to $O(n)$.

3. PARALLEL PROGRAMS

In the previous section, we mathematically defined a graph model for flow of control in a parallel computation based on token flow. This approach to understanding parallel computation has been used frequently in the past. Lester [19] presents a similar model. Lester [20] shows how a linear algebra technique similar to the one presented in section 2 can be used to analyze Petri Nets. In a survey paper on Petri Nets, Murata [21] gives a definition of *consistency* for Petri Nets, which is similar to our definition in section 2. Lee [22] shows how *consistency* of token flow can be used to locate structural errors in parallel dataflow graphs.

Table 1 Parallel program for histogram.

```
#define n 100    /*dimension of the image*/
#define max 20   /*maximum pixel intensity*/
int image[n][n]; /*image array*/
int hist[max+1]; /*computed histogram of image*/
spinlock L[max+1];  /*array of spinlocks*/
int i;

main( )  {
  ...
  for (i = 0; i <= max; i++)
    hist[i] = 0; /*initialize hist array*/
  forall i = 0 to n-1 do  {
    /*create one process for each row of image*/
    int j, intensity;
    for (j = 0; j < n; j++)  {
      intensity = image[i][j];
      Lock(L[intensity]);
        hist[intensity] = hist[intensity] + 1;
      Unlock(L[intensity]);
    }
 }
}
```

The original contribution in our current paper now to this research area, is the ability to apply this technique in an **efficient** and **automated** manner to real parallel programs. When one attempts to apply a graphical token flow technique to a real parallel program, one quickly finds that it is not obvious how to convert the various programming language features into an equivalent token flow model. Analyzing real parallel programs in this way is still an open research problem. We feel that our effort in this paper now has made a significant contribution in this endeavor.

In this paper, we report on our research in using control flow graphs to analyze parallel programs written in the language C*, a language developed mainly for education in parallel programming techniques (see *The Art of Parallel Programming* [23, 24]). The C* language consists of the C language with the addition of a few of the standard parallel programming primitives for creating parallel processes and exchanging messages between them. In this section and the next section, we describe the major features of the C* language, and techniques for creating equivalent control flow graphs.

To promote a clearer understanding, it will be helpful to begin with an informal example of a C* program and its corresponding control flow graph. *The Art of Parallel Programming* [24] contains a simple parallel program to compute the *histogram* of a visual image. This program is shown in Table 1. The image is represented by a two-dimensional array of integers, representing the light intensity at each point of the image. The intensity ranges from 0 up to some maximum value. The histogram displays the total number of pixels at each possible intensity value. The *forall* statement creates *n* parallel processes, one for each row of the image array. Each process scans its own row and increments the appropriate element in the histogram array *hist*. Since the elements of *hist* are incremented in parallel by different processes, each element of *hist* has its own spinlock to provide mutual exclusion during the increment operation.

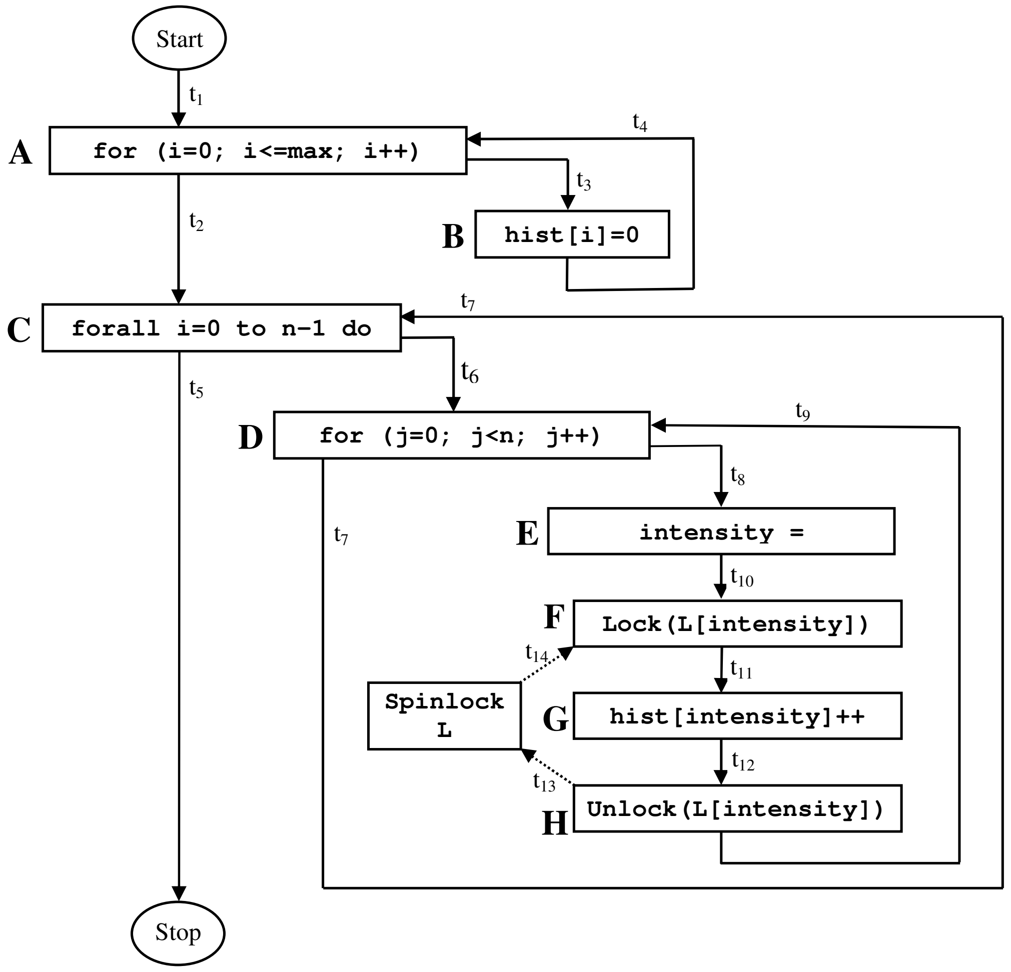

Figure 3. Control Flow Graph for Histogram Program.

The flow of control in this Parallel Histogram program can be represented by the control flow graph shown in Figure 3. Each statement in the original program becomes a module in the control flow graph. The terminals of the control flow graph are labeled t_1 to t_{14}. The array of spinlocks L is also represented as a single module in the control flow graph. In this and subsequent control flow graph examples, we sometimes use dashed lines to represent the terminals rather than solid lines. This is just to make the examples more understandable. With respect to the mathematical analysis, there is no difference between solid lines and dashed lines.

Looking at the first *for* statement labeled *A*, it is obvious that terminals t_1 and t_2 must have the same count for any properly terminating execution of the program. Similarly, terminals t_3 and t_4 must have the same count. Therefore, module *A* has the following linear equations:

$$[t_1] - [t_2] = 0$$

$$[t_3] - [t_4] = 0$$

At runtime, the execution of the program will determine the exact relationship between the counts $[t_1]$ and $[t_3]$. Since this relationship is not known at compile time, it is left unspecified in the above linear equations. Each of the other modules shown in Figure 3 also has its own linear equations governing the counts on its connecting terminals. Collecting all of these equations results in the system of control flow equations shown in Table 2. Using the standard techniques of linear algebra, these control flow equations have a general solution with four arbitrary constants (*a*, *b*, *c*, *d*):

$[t_1] = [t_2] = [t_5] = a$

$[t_3] = [t_4] = b$

$[t_6] = [t_7] = c$

$[t_8] = [t_9] = [t_{10}] = [t_{11}] = [t_{12}] = [t_{13}] = [t_{14}] = d$

According to the definition presented in section 2, the control flow graph of Figure 3 is therefore *consistent*. Now assume the programmer makes a slight error when typing the Histogram program of Table 1, and omits the *Unlock* statement. The resultant control flow graph will be similar to Figure 3, with module *H* and terminals t_{12} and t_{13} removed. This change will result in a small change to the control flow equations shown above. It is easy to show that the new control flow equations have only the trivial solution in which all terminal counts are 0. Thus, the control flow graph is not *consistent*. According to the Theorem presented in section 2, the control flow graph is therefore not *properly terminating*. Looking at Table 1, one can see that the removal of the *Unlock* statement does result in a program that is not properly terminating — the program will result in a deadlock.

Table 2 Control Flow Equations for Histogram Program.

Module	**Equations**
A	$[t_1] - [t_2] = 0$ $[t_3] - [t_4] = 0$
B	$[t_3] - [t_4] = 0$
C	$[t_2] - [t_5] = 0$ $[t_6] - [t_7] = 0$
D	$[t_6] - [t_7] = 0$ $[t_8] - [t_9] = 0$
E	$[t_8] - [t_{10}] = 0$
F	$[t_{10}] - [t_{11}] = 0$ $[t_{10}] - [t_{14}] = 0$
G	$[t_{11}] - [t_{12}] = 0$
H	$[t_{12}] - [t_9] = 0$ $[t_{12}] - [t_{13}] = 0$
Spinlock L	$[t_{13}] - [t_{14}] = 0$
Start	$[t_1] - [t_5] = 0$

4. C* LANGUAGE

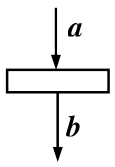

Figure 4. Module for Assignment Statement.

Now we will describe a practical technique for determining the control flow graph corresponding to any parallel program written in the C* language. The techniques described in this section are easily automated into a compiler, and form the basis of a prototype C* compiler developed by the author. Let us begin with a simple assignment statement. This can be represented as a module with one input terminal and one output terminal as shown in Figure 4.

When a virtual processor begins to execute this assignment statement, this can be represented in the control flow graph as the movement of a token through the input terminal *a*. This puts the module into the *active* state. When execution is complete, the token representing the virtual processor moves out of terminal *b*, returning the module to the *inactive* state. Therefore, whenever this assignment statement is in the *inactive* state, we know that the total control token count at terminal *a* exactly equals the token count at terminal *b*:

$[a] - [b] = 0$

For some programs, it may be possible to have multiple simultaneous executions of an assignment statement. In this case, the module will remain in the *active* state until all of these executions are complete. Therefore, when the module is in the *active* state, the count [*a*] may exceed the count [*b*], but in the *inactive* state, the above equality of [*a*] and [*b*] is always true. Thus, a linear equation of the above form is associated with every assignment statement in the C* program.

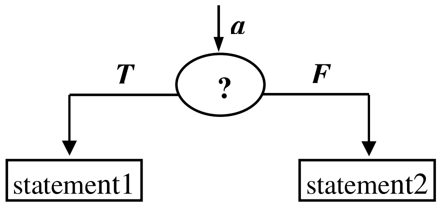

Figure 5. Module for Conditional Statement.

Now let us consider a conditional statement in C* of the form:

if (*condition*) *statement1*; **else** *statement2*;

When showing the general form of C* statements as in the above, we use bold to indicate keywords in the language. This conditional statement can be represented as a module with one input terminal and two output terminals as shown in Figure 5. The situation is similar to the assignment statement, except the virtual processor may exit through either the *T* or *F* output terminals. Thus, the conditional module will constrain the counts to satisfy the following linear equation whenever it is in the *inactive* state:

$[a] - ([T] + [F]) = 0$

Parallel processes in C* are created with a *forall* statement of the following general form:

forall i = *initial* **to** *final* **do** *statement*;

Let us first consider a *forall* statement in which the *initial* and *final* values are constants, and therefore known at compile time:

forall i = 1 **to** 3 **do** *statement*;

This *forall* can be represented by a module with one input terminal and three output terminals as shown in Figure 6.

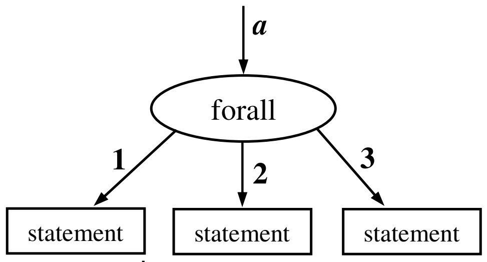

Figure 6. Forall Statement Module.

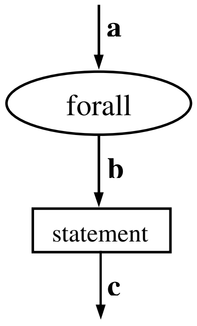

Figure 7. Forall Statement with Variable Range.

For each token entering through terminal a, this *forall* module outputs a token through each of the output terminals. When the module is in the *inactive* state, the count at each output terminal equals the count at terminal *a*, as expressed in the following three equations:

$[a] - [1] = 0 \quad [a] - [2] = 0 \quad [a] - [3] = 0$

The particular value of the *forall* index variable i (in this case, 1, 2, or 3) is substituted for i as a constant in the statement body of each process. Since a process often uses its own *forall* index extensively in control statements, these three parallel processes may result in control flow graphs that are considerably different from each other.

Now consider a *forall* statement in which the initial or final value is a program variable, not known at compile time:

forall i = 1 **to** n **do** *statement*

The corresponding module is shown in Figure 7. Since the control flow analysis of the program is done at compile time, we do not know how many parallel processes will be created. Therefore, the linear equations for this module do not constrain the count at terminal *b*. However, we do know that each token entering terminal *a* will eventually produce a token at terminal *c* after all modules have re-entered the *inactive* state. Thus, we have the following linear equation for this *forall* module:

$[a] - [c] = 0$

These two different representations of the *forall* statement illustrate the general approach we have used to create the control flow graph from the C* program. We use whatever information is available at compile time, to create a more accurate model of the behavior of the program. This will allow the consistency analysis of the resultant control flow graph to detect a wider range of control flow errors. However, if some information is not known at compile time, we can still proceed with the analysis using whatever information we do have.

5. FUNCTIONS AND COMMUNICATION STREAMS

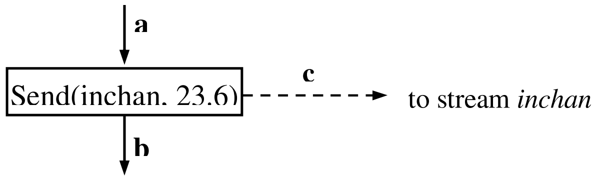

Figure 8. Module for Send Operation.

The C* parallel programming language has *shared variables* that are accessible to all processes. The language also has *stream variables* for message passing between processes. Stream variables are a separate data type and must be explicitly declared:

```
float stream inchan;     /* inchan is stream of float values */
float  myval;
```

To write a number into a stream in C* use the *Send* operation. The stream acts as a FIFO queue of numbers with new values being added at the end:

```
Send(inchan,  23.6);     /* write 23.6 into stream */
```

To read a value from the stream, use the *Recv* operation, as in the following example that reads a single value from stream *inchan* into variable *myval*:

```
Recv(inchan, myval);
```

If the stream *inchan* is empty when the above *Recv* operation is executed, the execution of the process will be suspended until another process puts a value into the stream. In this way, it is possible to use streams for interprocess communication and synchronization. We also see that streams introduce the possibility of circular waits and deadlocks if improperly used.

A *Send* operation statement can be represented as a module with one input terminal and two output terminals as shown in Figure 8. The flow of control equations are similar to a simple assignment statement, with an additional output terminal:

$$[a] - [b] = 0$$

$$[a] - [c] = 0$$

A *Recv* operation statement results in the same module as the *Send*, except the terminal *c* goes in the opposite direction.

The stream itself must also be represented as a separate module in the control flow graph. The *Send* operations to this stream form the input terminals of the module, and the *Recv* operations

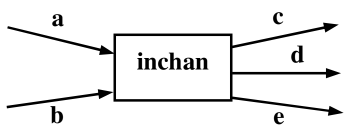

Figure 9. Module for Stream.

form the output terminals. For most programs, each steam will have a relatively small number of terminals. However, it is possible for some stream to have a very large number of terminals. Figure 9 shows an example module corresponding to a stream.

For a stream module, each data value written into the module is represented as a token that flows in through one of the input terminals. Reading a value from the stream is represented as the flow of a token out of one of the output terminals. A stream module is considered to be in the *inactive* state whenever it is empty. In the C* language, all streams are initially empty. For every individual stream module, the control flow equation is simply: the sum of the input flows equals the sum of the output flows. For the above stream module, this is as follows:

$$([a] + [b]) - ([c] + [d] + [e]) = 0$$

C* also allows an array of streams. If all references to a particular stream array have indexes known at compile time, then each element of the array is represented as a separate module. Otherwise, the whole array of streams is treated as a single module. This choice is automatically made for each stream at compile time while the control flow graph is being created.

Another important control structure in the C* parallel programming language is the for loop:

for (*expression1* ; *expression2* ; *expression3*) *statement*;

The corresponding module in the control flow graph is shown in Figure 10. Independent of the particular details of the three controlling expressions, we always know that the control flow into terminal *a* is equal to the control flow out of terminal *b*. Similarly, the flow out of terminal *c*

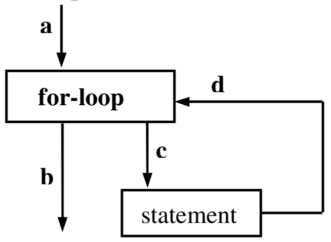

Figure 10. Module of *For* Loop.

equals the flow into terminal *d*:

$$[a] - [b] = 0 \qquad [c] - [d] = 0$$

C* program loops created using while statements or *do-while* statements will also have a similar control flow graph with four terminals. Since the number of iterations of the loop is not known at compile time, the above two equations capture all the information that is known at compile time.

Now let us consider C* function calls. We have two different techniques for converting functions into control flow graphs, depending on whether the function is recursive. For each call to a non-recursive function, the function call is expanded *inline* using the source code of the function body. This individual expansion is necessary because parallel processes often pass their own *forall* index value as a parameter to the functions. Therefore, the same function call appearing in different processes may expand into different control flow graphs.

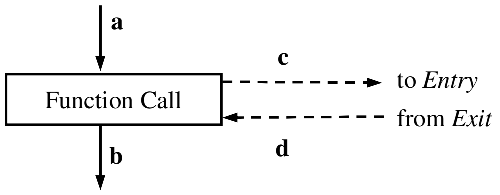

Figure 11. Module for Function Call.

This inline expansion technique will obviously not work for a call to a recursive function. We expand a recursive function into a control flow graph only once. A special *Entry* module is added to the start of the function, which sums all input control flows. A special *Exit* module is added to the end of the function, which sums all output control flows. Each call to the recursive function creates the module type shown in Figure 11. The control flow equations for this function call are as follows:

$[a] - [b] = 0 \quad [a] - [c] = 0 \quad [a] - [d] = 0$

In the C* language, a statement may be spawned into a separate parallel process using the *Fork* operator:

Fork *statement*;

This can be represented in the control flow graph with a *Fork* module (see Figure 12). The control flow equations for this *Fork* module are as follows:

$[a] - [b] = 0$

$[a] - [c] = 0$

One final module to consider is the *START* module. In the control flow graph, the output terminal of the *START* module is connected into the input terminal of the first statement of the main function body of the C* program. The output terminal of the last statement of the main function body is connected to the input terminal of the *START* module. The control flow equation for the *START* module is simply: the count at the input terminal equals the count at the output terminal. In the informal example of a control flow graph shown in Figure 3, a separate *Stop* module is shown to make the Figure easier to understand. However, strictly speaking the

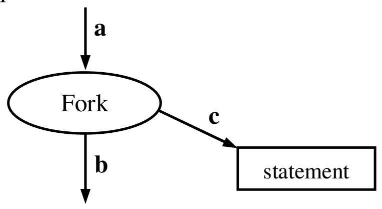

Figure 12. Module for *Fork* Operation.

Stop module should be eliminated, and the terminal t_5 should terminate on the *Start* module.

As we can see from the modules and equations described in this section for the C* programming language, the control flow matrix will be very sparse. Almost all of the rows and columns will have only two or three non-zero elements. The number of rows and columns will be a small multiple of *n*, the number of statements in the program. If the control flow matrix is represented as an ordinary two-dimensional array, consistency can be determined using standard Gaussian Elimination in time $O(n^3)$.

This execution time can be considerably reduced by using a standard sparse matrix representation: a linked list for each row containing only the non-zero entries, and similarly for each column. In Gaussian Elimination, there are three nested loops, each with $O(n)$ repetitions—hence the $O(n^3)$ overall time complexity. However, the two inner loops only operate on non-zero entries. Therefore, this sparse matrix representation can be used to reduce the time complexity of the two inner loops to a constant independent of *n*. Thus, the overall time for determining *consistency* through Gaussian Elimination is reduced to $O(n)$.

6. EXAMPLE: PARALLEL JACOBI RELAXATION

In our research, we have developed techniques for creating a corresponding module and control flow equations for all of the statements in the C* language. However, due to lack of space in this short paper, we have not presented all these statements in detail. To test the validity and practical application of our technique, we modified an existing compiler for the C* parallel programming language to include a control flow consistency test, as defined in section 2. We began with a C* compiler originally developed to accompany the textbook *The Art of Parallel Programming* [24]. It is a recursive descent compiler consisting of a single pass through the C* source code. The output of this pass is an executable program consisting of a series of pseudo-code instructions of the following form:

operator *operand1*, *operand2*

There are a total of 115 different pseudo-code operators, most of which are standard machine language operations such as add, multiply, branch, store, and load. However, there are also some higher level operations, such as create a process, request a lock, or read a value from a stream.

To perform a control flow *consistency* test, we added a second pass to the compiler that scans through the pseudo-code program and builds the control flow matrix *F*. After completing this second pass, the compiler then submits the matrix *F* to a linear equation solver that uses standard Gaussian Elimination to solve the system of equations $F\,\mathbf{x} = \mathbf{0}$. If a solution exists for which all $x_i \neq 0$, then the C* program is *consistent*. Otherwise, the compiler issues a warning message to the programmer that the program has a control flow error that will result in an improper termination.

Now let us consider a brief example: a parallel program to solve Laplace's Equation using the Jacobi Relaxation algorithm. Consider a simple application problem to compute the voltage distribution on the surface of a two-dimensional conducting metal sheet when a known voltage is applied along the four boundaries. The voltage distribution $v(x, y)$ on the metal sheet can be computed by solving Laplace's Equation:

$$\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} = 0$$

This equation can be solved numerically by considering a two-dimensional grid of points on the

Table 3 Parallel Jabobi Relaxation program.

```
#define n 100          /* size of array */
#define numiter 1000   /* number of iterations */
float A[n+2][n+2];     /* array of points */
float stream C[n+2][n+2]; /*array of streams: one for each process*/
int i, j;
main( ) {
  /* create processes to hold boundary points constant */
  fork LeftBoundary( );
  fork RightBoundary( );
  fork TopBoundary( );
  fork BottomBoundary( );
  /* create processes for interior points */
  forall  i = 1 to n do
    forall  j = 1 to n do {
      int m;   /* local variable*/
      float sum,inval;  /* local variables */
        for (m = 1; m <= numiter; m++)  {   /* 1000 iterations */
          /* send my point to four neighbors */
          Send(C[i][j-1], A[i][j]);   /* send left  */
          Send(C[i][j+1], A[i][j]);   /* send right */
          Send(C[i-1][j], A[i][j]);   /* send up    */
          Send(C[i+1][j], A[i][j]);   /* send down  */
          /* receive values sent by my four neighbors */
          Recv(C[i][j], inval);   sum = sum + inval;
          Recv(C[i][j], inval);   sum = sum + inval;
          Recv(C[i][j], inval);   sum = sum + inval;
          Recv(C[i][j], inval);   sum = sum + inval;
          A[i][j] = sum / 4.0;  /*average of neighboring points*/
        }
    }
}
```

surface of the metal sheet. The voltage at all the boundary points is known and held constant. Initially, the voltage at each internal point is arbitrarily set to 0. Then the voltage at each internal points is iteratively recomputed as the average of the four neighboring points: above, below, left, and right. This computational method is called *Jacobi Relaxation*. It is easily parallelized

Table 4 Boundary Processes for Jacobi Relaxation.

```
void LeftBoundary( ) {
  int k;
  forall k = 1 to n do {
      int m;  float z;
      for (m = 1; m <= numiter; m++)  {
        Send(C[k][1], A[k][0]);
        Recv(C[k][0], z);
      }
  }
}
void RightBoundary( ) {
   // ... not shown
}
void TopBoundary( ) {
  int k;
  forall k = 1 to n do {
      int m; float z;
      for (m = 1; m <= numiter; m++)  {
        Send(C[1][k], A[0][k]);
        Recv(C[0][k], z);
      }
  }
}
void BottomBoundary( ) {
   / ... not shown
}

main( ) {  /* main function shown in Table 3 */  }
```

by partitioning the points and assigning a different process to each partition. For simplicity, we will assign just a single point to each process.

The main data structure is a two-dimensional array of points *A*. Each of the points is assigned to a separate parallel process that iteratively recomputes the value of the point using the average of the four immediate neighboring points. The processes exchange values using an array of streams *C*, with one stream assigned to each process. The main body of the program is shown in Table 3. The nested *forall* statements create n^2 = 10,000 parallel processes. Process (*i*, *j*) is assigned point *A*[*i*][*j*] and uses stream *C*[*i*][*j*] to receive values from its four neighbors. Each process sends the current value of its own point to all four neighbors, then receives and computes the average of the four values sent by the neighbors. The points along the four boundaries are held constant by the special processes created by the four procedures (Table 4).

The control flow graph for this C* program has a total of 49 modules and 81 terminals. The corresponding control flow equations $F\,\boldsymbol{x} = \mathbf{0}$ do have a solution for which all $x_i \neq 0$. Therefore, the control flow graph is consistent. Now assume the programmer has made a slight error when writing the program and has omitted the "Send Left" instruction: Send(C[i][j-1], A[i][j]). This will disrupt the global data flow and eventually result in a massive deadlock of all processes during program execution. Using an ordinary compiler, this error will not be detected because the program does not have any syntax errors. However, using our new prototype compiler, the programmer will be informed that the control flow in the program is not *consistent* and therefore will result in an improper termination. Thus, this new analysis technique for parallel programs is capable of identifying program errors at compile-time that would otherwise not be noticed until the program is executed.

7. IDENTIFYING PROGRAM ERRORS

The previous sections of this paper have outlined a technique to create a corresponding control flow graph for any C* program, and then determine whether the control flow graph is *consistent* by solving the control flow equations. If the control flow graph is not *consistent*, what will happen when the C* program is executed? In this context, we will use the following new definitions with respect to a particular execution of a control flow graph:

(i) **Non-Termination**: control flow never reaches the input terminal of the *START* module, or

(ii) **Improper Termination**: when the control flow does reach the input terminal of the *START*, one or more modules is still in the *active* state.

A series of test executions E_1, E_2, …, E_k of a C* program is said to *cover* the program if every statement in the program is executed at least once during this series. Now consider such a series of test executions. According to the Theorem presented in section 2, if a control flow graph is not *consistent*, then it is not *properly terminating*. Therefore, at least one E_i in this series will result in either *non-termination* or *improper termination*.

Non-termination indicates that the program either has a deadlock or an infinite internal execution cycle, both of which are certainly pathological and should be reported to the programmer as control flow errors. To understand *improper termination*, we must analyze what it means for a module to still be in the *active* state after the program has reached its end. There are two basic categories of modules: *executable* and *non-executable*. The *executable* modules are used to represent the executable statements of the C* program, such as assignments, loop control statements, *forall* statements, function calls, etc. The *non-executable* modules are used to represent *stream* variables. An *executable* module is *active* if it is in the midst of an execution. A stream is *active* if it contains at least one unread message.

Using the C* language, it is not possible for control flow to reach the end of the program while some statement is still in the midst of an execution (see reference [24]). Therefore, *improper termination* indicates that the C* program will terminate with one or more unread messages in the streams. We claim that this should be considered as pathological, and reported to the programmer as a warning during program compilation.

However, some programmers may want the option of writing C* programs that terminate with some unread messages in the streams. The C* language allows this without generating a runtime error message. Therefore, it will be useful to have some mathematical technique to differentiate between *non-termination* and *improper termination*. The control flow equation for each stream module uses a sum property:

sum of the input terminal counts – sum of the output terminal counts = 0

To allow unread messages in a stream, we introduce an additional *slack* variable x_s representing the unread messages in the stream. The modified control flow equation is as follows:

sum of the input terminal counts – sum of the output terminal counts – x_s = 0

(subject to the positivity constraint $x_s \geq 0$)

Furthermore, the stream module is always in the *inactive* state, which means this equation is true at all times. (This corresponds to the defined behavior of streams in the C* language.) The control flow equations for the non-stream modules remain the same as before. Collecting these equations together, we get the following modified system of control flow equations:

$$G\,\mathbf{x} = \mathbf{0}$$
$$\mathbf{x} \geq \mathbf{0}$$

Assume that x_n is the variable corresponding to the count at the input terminal of the *START* module, which corresponds to termination of the program. A solution to the above equations

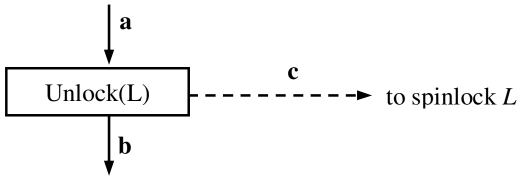

Figure 13. Module for Unlock Operation.

can be found using the following linear programming problem:

minimize x_n

subject to $G\boldsymbol{x} = \mathbf{0}$ (control flow equations)

$\boldsymbol{x} \geq \mathbf{0}$ (positivity constraint)

$x_j \geq 1$, for all terminal count variables

A control flow graph is said to be *weakly consistent* if the above linear programming problem has a feasible solution (a feasible solution is one that satisfies all the constraints, but does not necessarily optimize the objective function). A linear programming problem can be solved using the standard simplex algorithm, the first step of which is to find a feasible solution, or report that no feasible solution exists [25]. Although linear programming can be exponential in the worst case, the simplex algorithm almost always requires only polynomial execution time.

Theorem: Let E_1, E_2, …, E_k be series of test executions that *cover* a given C* program. If the corresponding control flow graph is not *weakly consistent*, then at least one E_i in this series will result in *non-termination*.

Proof: The proof follows easily from the definitions. We use proof by contradiction. Assume the control flow graph is not *weakly consistent*, but all the test executions in the series are terminating. Using the modified control flow equations for stream modules, an *improper termination* is not possible for a control flow graph derived from a C* program. Therefore, each execution E_j in the series results in a proper termination. Thus, the terminal counts for each E_j provide a particular integral solution to the modified control flow equations $G\,\boldsymbol{x} = \mathbf{0}$, where $\boldsymbol{x} \geq \mathbf{0}$. To construct this solution, use $x_p = [p]$ for each terminal variable x_p; and for each stream variable s, set the corresponding slack variable x_s to the number of unread messages remaining in the stream at the end of the execution E_j.

Since the series of executions cover the C* program, each specific terminal t has an associated execution E_j in the series such that $[t] \geq 0$, and therefore $[t] \geq 1$. This provides a solution to $G\boldsymbol{x} = \mathbf{0}$, in which $\boldsymbol{x} \geq \mathbf{0}$ and $x_t \geq 1$. We simply sum these solutions across all the terminals to get another solution to $G\,\boldsymbol{x} = \mathbf{0}$ in which $\boldsymbol{x} \geq \mathbf{0}$ and $x_t \geq 1$ for all terminals t. Therefore, the control flow graph is *weakly consistent*. This contradicts the original assumption that the control flow graph is not *weakly consistent*.

(End of Proof)

We now have established two different properties of a C* program that can be automatically tested at compile time: *consistency* and *weak consistency*. If the program fails the *consistency* test, then any series of test executions that cover the program will contain an execution with one of the following types of control flow errors:

- Deadlock
- Infinite internal execution cycle
- Unread messages in the streams at termination

If the program fails the *weak consistency* test, then at least one execution in any covering series will result in deadlock or an infinite internal execution cycle.

8. LOCKING OF SHARED DATA

In addition to communication using stream variables, the C* language also allows parallel processes to interact by reading and modifying shared variables. Therefore, some mechanism is required to provide mutual exclusion during modification of the shared variables. This is done using a special *spinlock* data type. A *spinlock* variable has two states: *locked* and *unlocked*. The initial state is *unlocked*. The *Lock*( ) operation changes the state from *unlocked* to *locked*. The *Unlock*( ) operation changes the state from *locked* to *unlocked*. Consider the following portion of a C* program:

```
spinlock L;  /* declaration of L as spinlock variable */
Lock(L);
   . . .    /* atomic modification to shared data */
Unlock(L);
```

Frequently, the *Lock* and *Unlock* operations are used in matched pairs as shown above. However, this is not required in C*. *The Art of Parallel Programming* [24] contains several examples in Chapter 5 of programs in which *Lock* and *Unlock* operations are used in more complex ways. Bugs in the use of these operations can create program deadlocks. Therefore, it will be useful to include spinlocks in the control flow graph when determining *consistency* or *weak consistency*. As was illustrated in Figure 3, each spinlock variable can be represented in the control flow graph as a single module. An *Unlock* operation is represented with the module shown in Figure 13. The flow of control equations are as follows:

$[a]-[b] = 0 \qquad [a]-[c] = 0$

A *Lock*(*L*) operation results in the same module as *Unlock*, except terminal *c* goes in the opposite direction. Now let us consider the control flow equation for the spinlock module itself. For consistency testing, we need a linear equality. One possibility is to use a sum property as was done for the stream modules: the sum of the inputs equals the sum of the outputs. Since the inputs are from *Unlock* operations, and the outputs go to *Lock* operations, this equation requires that the number of *Lock* operations on each spinlock exactly equals the number of *Unlock* operations. We can include this in the control flow equations used to determine *consistency* of the control flow graph. Therefore, an imbalance in the number of *Lock* and *Unlock* operations will be flagged as a control flow error.

However, the C* language does not actually require that *Lock* and *Unlock* operations are exactly balanced. There can be one extra *Lock* operation, which will cause the spinlock to be in the locked state at program termination. There can also be any number of extra *Unlock* operations. After a spinlock enters the *unlocked* state, additional *Unlock* operations are just ignored. To allow this more liberal use of *Lock* and *Unlock* operations, the following **linear inequality** can be used as the control flow equation for each spinlock module:

sum of input terminal counts – sum of output terminal counts $\geq$ -1

Since the control flow equations for testing *weak consistency* allow such linear inequalities, this new more liberal model of spinlocks can be used in the *weak consistency* test. Although we will not prove it formally, it is fairly easy to show that the Theorem proved in the previous section still holds: if the control flow graph of a C* program is not *weakly consistent*, then any covering series of test executions will contain at least one non-terminating execution. Thus, *weak consistency* testing is still a very useful method of finding control flow errors in C* programs that use spinlocks and/or streams. We have already seen that the Parallel Histogram program of Table 1 is not *consistent* if the *Unlock* operation is removed. It also is not *weakly consistent*, and therefore should be non-terminating. We have seen that removal of the *Unlock* results in a deadlock, which is included in our definition of non-termination.

9. CONCLUSIONS AND FUTURE RESEARCH

In this paper, we have presented a simple and practical technique for analyzing the control flow properties of parallel programs at compile time. This technique is computationally efficient, and does not suffer from some of the limitations associated with previous research in automated program analysis. The next step in this research is to apply this technique to other parallel programming languages besides C*.

Although we have compared our technique to automated verification techniques, strictly speaking it is not really an automated verification technique. Automated *error-finding* technique would perhaps be a better description. If the parallel program does pass the consistency test (or weak consistency test), this does not guarantee that the program will terminate properly. However, if the program fails the *consistency* test (or *weak consistency* test), then the program definitely has a control flow error. Therefore, testing these properties at compile time can save the programmer a great deal of work during program debugging.

Compilers already perform some limited error-checking of programs. For example, if the programmer makes a typo and misspells a variable name, the compiler will flag it as an undefined identifier. Although this limited error-checking does not guarantee the program will work correctly, it is nevertheless very helpful to the programmer. The same is the case for the technique of control flow analysis presented in this paper. By identifying control flow errors at compile time, the effort required for program debugging is reduced.

Using linear equations to analyze token flow models of parallel computation is not something new. As summarized at the beginning of this paper, there has been a great deal of past research in this area. However, the unique research contribution of this paper has been to apply this technique in a practical and efficient way to a real parallel programming language. We already have a working compiler for the C* language that determines whether the program is *consistent*. We are currently modifying this compiler to also perform a *weak consistency* test. As the next step in this research, we would like to apply this technique to MPI and OpenMP.

REFERENCES

[1] G.S. Avrunin, et al., "Automated analysis of concurrent systems with the constrained expression toolset," IEEE Transactions on Software Engineering, 17(11), pp. 1204-1222, 1991.

[2] G.S. Avrunin, et al., "Finite-state verification for high performance computing," in Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications, ACM Press, New York, NY, 2005, pp. 68-73.

[3] P. Godefroid, P. Wolper, "Using partial orders for the efficient verification of deadlock freedom and safety properties," Formal Methods in System Design, 2(2), pp. 149-164, 1993.

[4] G.J. Holzmann, The SPIN Model Checker, Addison-Wesley: Boston, MA, 2004.

[5] D.P. Helmbold, C.E. McDowell, "Computing reachable states of parallel programs," in: Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, California, 1991, pp. 76-84.

[6] A. Valmari, "A stubborn attack on state explosion," in: Proceedings 2nd Int. Workshop on Computer-Aided Verification, LNCS 531, Springer-Verlag, New York, NY, 1991, pp. 156-165.

[7] G. S. Avrunin, et al., "Comparing Finite-State Verification Techniques for Concurrent Software," Technical Report UM-CS-1999-069, Department of Computer Science, University of Massachusetts, 1999.

[8] S. P. Masticola , B. G. Ryder, "A model of Ada programs for static deadlock detection in polynomial times," in: Proceedings of the 1991 ACM/ONR workshop on parallel and distributed debugging, Santa Cruz, California, 1991, pp. 97-107.

[9] S. Shatz, et al, "An application of Petri Net reduction for Ada tasking deadlock analysis," IEEE Transcations on Parallel and Distributed Systems, 7(12), pp. 1307-1322, 1996.

[10] B. Schaeli and R. D. Hersch, "Dynamic testing of flow graph based parallel applications, " in: Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging, Seattle, Washington, 2008, pp. 1-10.

[11] G. K. Baah, et al., "The probabilistic program dependence graph and its application to fault diagnosis, " in: Proceedings of the 2008 international symposium on Software testing and analysis, Seattle, Washington, 2008, pp. 189-200.

[12] B. Elkarablieh, D. Marinov,and S. Khurshid, "Efficient solving of structural constraints, " in: Proceedings of the 2008 international symposium on Software testing and analysis, Washington, 2008, pp. 39-50.

[13] B. Schaeli and R. D. Hersch, "Dynamic testing of flow graph based parallel applications, " in: Proceedings of the 2008 international symposium on Software testing and analysis, Seattle, Washington, 2008, pp. 155-166.

[14] C. Tian, et al., "Dynamic testing of flow graph based parallel applications, " in: Proceedings of the 2008 international symposium on Software testing and analysis, Seattle, Washington, 2008, pp. 143-154.

[15] J. Chen and S. MacDonald, "Towards a better collaboration of static and dynamic analyses for testing concurrent programs, " in: Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging, Seattle, Washington, 2008.

[16] B. P. Lester, "Automatic detection of control flow errors in parallel programs," in Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, MA, 2004, pp. 368-373.

[17] B. Kolman, D.R. Hill, Elementary Linear Algebra, 7/E, Prentice Hall: Upper Saddle River, NJ, 2000.

[18] T.A. Davis, Direct Methods for Sparse Linear Systems, SIAM: Philadelphia, PA, 2006.

[19] B.P. Lester, "Coherent flow of information in parallel systems," in: Proceedings of 1983 International Conference on Parallel Processing, Chicago, Illinois, 1983.

[20] B.P. Lester, "Analysis of firing rates in Petri Nets using linear algebra," in: Proceedings of 1985 International Conference on Parallel Processing, Chicago, Illinois, 1985, pp. 217-224.

[21] T. Murata. "Petri nets: properties, analysis and applications," Proceedings of the IEEE, 77(4), pp. 451-480, 1989.

[21] E. A. Lee, "Consistency in dataflow graphs," IEEE Transactions on Parallel and Distributed Systems, 2(2), pp. 223-235, 1991.

[23] B.P. Lester, The Art of Parallel Programming, First Edition, Prentice Hall: Englewood Cliffs, NJ, 1993.

[24] B.P. Lester, The Art of Parallel Programming, Second Edition, 1st World Publishing: Fairfield, Iowa, 2006.

[25] S.I. Gass, Linear programming: methods and applications, fifth edition, Dover Publications, Mineola, NY, 2003.

Author

Dr. Bruce Lester received his Ph.D. in Computer Science from M.I.T. in 1974. He was a Lecturer in the Department of Electrical Engineering and Computer Science at Princeton University for two years. During 1979-1989, Dr. Lester was a faculty member at Maharishi University of Management (MUM) and served as Chair of the Computer Science Department. Beginning in 1989, Dr. Lester worked independently as an author and software engineering contractor. In 2007, he rejoined the faculty of Maharishi University of Management, where he is currently Professor of Computer Science.